



Processing

Introduzione a Processing

di MATTEO FIORAVANTI

Prosegue il nostro corso su Processing: in questa puntata vedremo come costruire un'interfaccia grafica e in che modo collegarla con Arduino. Seconda puntata.

Processing e Arduino sono due realtà molto simili, infatti il loro editor (o IDE) è praticamente lo stesso e la sintassi da utilizzare per programmare cambia davvero poco tra i due. L'unica differenza significativa è che il loro campo di applicazione è differente, in quanto Processing serve per scrivere software da PC, mentre l'IDE di Arduino ci consente di

programmare il microcontrollore che si trova a bordo proprio dei moduli Arduino. Qualunque sia il punto di partenza, è comunque molto semplice passare da un ambiente all'altro: da Arduino a Processing o viceversa. Questo corso riguarda Processing, quindi rimandiamo chi volesse approfondire la conoscenza di Arduino all'apposito corso; tuttavia in questa lezione

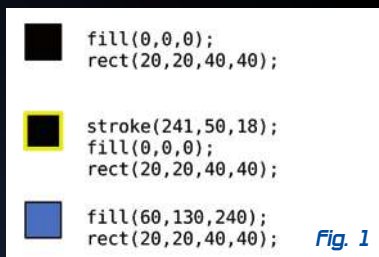


Fig. 1

ne coinvolgeremo la piattaforma Arduino perché intendiamo utilizzarla per testare i programmi d'esempio relativi agli esercizi che vi proponiamo.

Per questa ragione riportiamo in queste pagine anche il codice per Arduino ed il circuito di test corrispondente.

COMUNICAZIONE SERIALE

L'anello di congiunzione tra un programma scritto in Processing ed uno che gira su Arduino è la comunicazione seriale, la quale fisicamente è realizzata attraverso il cavo USB tramite cui Arduino è collegato al PC. A livello logico, nella comunicazione seriale vengono scambiati byte; ricordiamo che un byte è un numero binario di 8 bit, perciò la rappresentazione di un byte può essere un numero intero che va da 0 a 255 oppure la rappresentazione di caratteri che si trovano nella tabella ASCII. Che sia un numero, oppure un carattere, tutto dipende da come interpretiamo i dati; la sostanza comunque non cambia, perché si tratta sempre di un pacchetto di 8 bit che viene scambiato. Concettualmente una comunicazione è definita quando conosciamo il supporto fisico (nel nostro caso è il collegamento USB), il livello logico (cioè come sono interpretati i bit) ed infine, quando è stabilito il protocollo di comunicazione. Nel caso di questo corso, saremo noi a definire il protocollo, cioè attribuiremo a determinati simboli e numeri delle funzionalità, sia verso Processing che verso Arduino.

Una comunicazione deve essere opportunamente gestita nelle sue fasi; solitamente viene definito "master" il sistema che invia comandi con una certa cadenza temporale oppure a seguito di determinati eventi, mentre è chiamato "slave" il sistema che risponde alle interrogazioni del master. In questa lezione stabiliremo che il software da PC scritto in Processing sarà il master della comunicazione, mentre Arduino sarà lo slave. Possiamo impostare tutto il sistema anche con Arduino che funziona da master,



Fig. 2

ma lasceremo ciò come esercizio. A questo livello non ci sono particolari differenze, né vantaggi ad utilizzare una soluzione anziché l'altra.

Richiamati questi brevi concetti base, andremo adesso ad analizzare i codici oggetto di questa puntata del corso.

IL CODICE

I programmi di esempio che vogliamo proporvi stavolta sono i seguenti.

1. LEZIONE 2_1 BUTTON – Questo programma mostra come disegnare un pulsante, che si evidenzia quando il puntatore è sopra di esso e si accende se viene fatto clic.
2. LEZIONE 2_2 BUTTON & LED – Rispetto al programma precedente, aggiunge un LED che lampeggia in maniera intermittente ogni secondo.
3. LEZIONE 2_3 BUTTON & LED + TRASMISSIONE + ARDUINO – A questo punto utilizziamo Arduino; l'interfaccia è quella del programma precedente, ma ad ogni secondo viene trasmesso un comando ad Arduino, che accenderà o meno un LED secondo lo stato del pulsante grafico.
4. LEZIONE 2_4 BUTTON & LED + TRASMISSIONE/RICEZIONE + ARDUINO – Aggiungiamo ora la possibilità di leggere lo stato di ingresso di Arduino; in pratica ad ogni secondo viene inviato, da parte di Processing, lo stato del pulsante grafico ed Arduino risponde comunicando lo stato del suo pulsante reale. Processing accenderà un ulteriore LED sull'interfaccia grafica.

LEZIONE 2_1 BUTTON

Iniziamo ora a scrivere il nostro primo programma di questa lezione, cioè l'implementazione di un pulsante che si evidenzia se il mouse ci passa sopra e si illumina se viene fatto clic sopra di esso, quindi si spegne se facciamo nuovamente clic.

La strategia è relativamente semplice:

innanzitutto un pulsante può essere rappresentato come un quadrato riempito di colore nero, perciò evidenziare il rettangolo significa ridisegnare il quadrato con il bordo giallo, illuminare il pulsante equivale a ridisegnare il quadrato con il riempimento diverso da nero (magari un azzurro molto luminoso) come rappresentato in Fig. 1. Abbiamo quindi la necessità di gestire le primitive grafiche `rect()`, `fill()`, `stroke()`; inoltre ci serve una funzione che chiameremo `overRect()`, la quale ci restituirà un valore logico `TRUE` se il puntatore del mouse sarà sopra il rettangolo identificato come pulsante, secondo quanto schematizzato in Fig. 2. Infine ci occorre una piccola astuzia: definiamo innanzitutto la variabile `button` come intero; questa variabile sarà rappresentativa dello stato del pulsante. Poi scriveremo una funzione di riempimento in questo modo:

```
fill(60*button,130*button,
240*button).
```

Quando `button` vale 0, avremo questa funzione di riempimento: `fill(0,0,0)`, che significa riempimento di colore nero. Se `button = 1`, allora si ha `fill(60,130,240)`, che in questo caso corrisponde al riempimento con un colore RGB

Listato 1

```
int button; //definizione della variabile button come intero

void setup(){
  button = 0; //inizializzazione di button

  //definizione delle primitive grafiche
  color(255,255,255); //tipo di colore
  strokeWeight(1); //spessore linee
  strokeJoin(SQUARE); //tipo di spigolo: quadrato

  fill(100); //grigio di sfondo
  smooth();
  size(300,160); //area dell'applicazione

  fill(220,220,220); //grigio dell'area pulsante
  rect(10,10,280,140); //area pulsante

  textSize(30); //dimensione del testo
  fill(0);
}

void draw(){
  fill(60*button,130*button,240*button); //il colore di riempimento del pulsante
  //dipende dalla variabile button, ovvero dallo stato di button,
  //se button=0 ---> fill(0,0,0) quindi colore nero
  //Se button=1 ---> fill(60,130,240) quindi colore blu

  rect(20,40,40,40); //rettangolo che rappresenta il pulsante
  text("button", 20, 120); //etichetta

  strokeWeight(3); //spessore del bordo
  if (overRect(20,40,40,40)) //la funzione overRect restituisce TRUE se
  //il mouse è sopra il rettangolo che rappresenta il pulsante, altrimenti FALSE

  stroke(241, 250, 18); //definizione di una linea di colore giallo
  rect(20,40,40,40); //disegno di un rettangolo di colore giallo in
  //corrispondenza del pulsante, questa tecnica serve ad evidenziare il pulsante
}
else
{
  stroke(0, 0, 0); //se il mouse non si trova sopra il pulsante, il bordo
  //del pulsante è di colore nero
  rect(20,40,40,40);
}

//cambio dello stato del pulsante se tasto mouse premuto
if ((overRect(20,40,40,40)) && mousePressed) //se contemporaneamente il
//mouse è sopra l'area del pulsante ed il tasto sinistro è premuto viene cambiato
//il valore di button in 0 se era 1, oppure in 1 se precedentemente era 0
{
  delay(300); //questo ciclo di ritardo funziona da debounce del tasto del
  //mouse
  if (button == 0)
  {
    button = 1;
  }
  else
  {
    button = 0;
  }
}

boolean overRect(int x, int y, int w, int h) //funzione overRect, vengono
//passati i parametri di posizione X e Y, e di larghezza W e altezza H del pulsante
{
  if (mouseX >= x && mouseX <= x+w && mouseY >= y && mouseY <= y+h) {
    //overRect assume valore TRUE se il cursore del mouse è dentro il rettangolo
    //posizionato in X,Y e di dimensioni W,H
    return true;
  }
  else {
    return false;
  }
}
```


Listato 2

```

int button;           //definizione della variabile button come intero
int LED;              //definizione della variabile LED come intero
float lastUpdate;

void setup(){
  button = 0;          //inizializzazione di button
  LED = 0;             //inizializzazione di LED
  lastUpdate = 0;      //inizializzazione di lastUpdate

  //definizione delle primitive grafiche
  color(RGB);          //tipo di colore
  strokeWeight(1);     //spessore linee
  strokeJoin(SQUARE);  //tipo di spigolo: quadrato

  fill(30);            //grigio di sfondo
  smooth();
  size(300,360);       //area dell'applicazione

  fill(220,220,220);   //grigio dell'area pulsante
  rect(10,10,280,140); //area pulsante
  rect(10,160,280,140); //area LED

  textSize(30);        //dimensione del testo
  fill(0);
}

```

primitiva di Processing chiamata **mousePressed** cambiamo il valore di **button** da 0 a 1 e con esso il colore di riempimento. Il codice corrispondente all'applicazione qui descritta è contenuto nel Listato 1. Analizziamo adesso, più approfonditamente, la funzione **overRect()**. Diciamo innanzitutto che le funzioni si scrivono in fondo al programma principale; in

definito, nello specifico, dai valori 60, 130, 240 delle componenti cromatiche, rispettivamente, R, G, B.

Lo stesso principio descritto per **fill()** vale anche per **stroke()**, cioè per il bordo del rettangolo. Quindi, attraverso la funzione **overRect()** identifichiamo se il puntatore è sopra il pulsante; se, contemporaneamente, il tasto del mouse è premuto, attraverso la

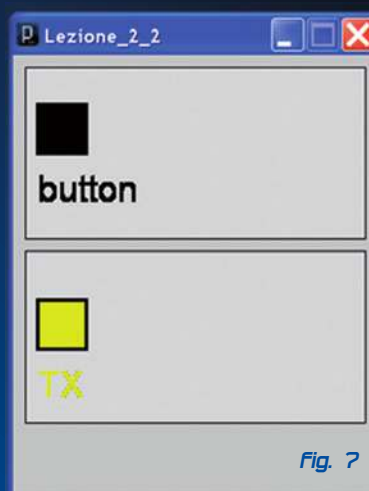
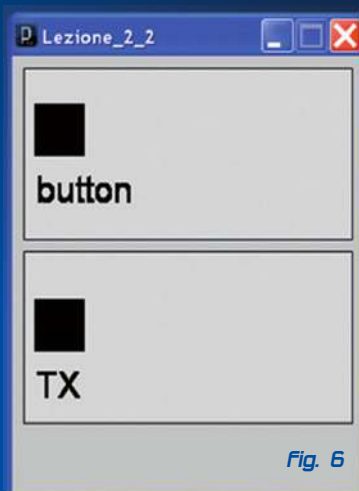
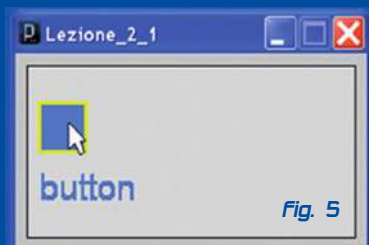
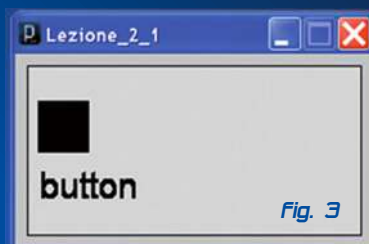
pratica fuori da **void draw()**. La funzione in questione è definita in questo modo:

```
boolean overRect(int x, int y, int w, int h){}
```

Ciò significa che **overRect** restituisce un valore logico che può essere **TRUE** o **FALSE** e si aspetta, come parametri di ingresso, quattro valori interi **x**, **y**, **w**, e **h**.

In sostanza, quando all'interno del programma richiamiamo questa funzione, dobbiamo passare la posizione **x**, **y** del rettangolo e le sue dimensioni **w**, **h**; la funzione restituirà **TRUE** o **FALSE**.

La verifica della posizione del puntatore del mouse, all'interno di **overRect**, viene svolta in questo modo:



Listato 3

```

void draw(){
...

//disegno del Led
strokeWeight(3);
stroke(0, 0, 0);
fill(241*LED,250*LED,18*LED); //il colore di riempimento del Led dipende dalla variabile LED, ovvero dal
dal suo stato
                                //se LED=0 ---> fill(0,0,0) quindi colore nero
                                //se LED=1 ---> fill(241,250, 18) quindi colore giallo
rect(20,200,40,40); //rettangolo che rappresenta il Led
text("TX", 20, 280); //etichetta

...
}

```

```

if (mouseX >= x && mouseX <= x+w &&
mouseY >= y && mouseY <= y+h) {
  return true;
}else{
  return false;
}

```

Viene, cioè, verificata una condizione con quattro AND logiche; il valore X del puntatore deve essere maggiore della posizione x del rettangolo e minore della posizione x+w (larghezza del rettangolo), mentre il valore Y del puntatore deve essere maggiore della posizione y del rettangolo e minore di y+h (altezza del rettangolo). Il risultato finale di questa prima interfaccia grafica è rappresentato nella Fig. 3, dove il pulsante è spento, in Fig. 4, che mostra il pulsante evidenziato e nella Fig. 5, dove il pulsante è acceso.

LEZIONE 2_2 BUTTON & LED

Aggiungiamo al programma precedente il comando di un LED (Listato 2). Anche in questo caso rappresentiamo il LED con un rettangolo colorato di nero se il LED è spento, ovvero di giallo se il LED è acceso. Innanzitutto dobbiamo cambiare l'interfaccia grafica del programma precedente, estendendo l'area del programma ed aggiungendo un nuovo rettangolo, con sotto l'etichetta 'TX'; il risultato si vede nella Fig. 6.

Oltre a dover ridisegnare l'area dell'interfaccia, abbiamo bisogno di una nuova variabile che chiameremo LED; inoltre ci servirà ancora un'altra variabile che chiameremo lastUpdate e della quale analizzeremo tra poco la funzionalità. Anche per il LED useremo la stessa tecnica sviluppata per il pulsante, ovvero agiremo sulla funzione di riempimento fill() e la renderemo dipendente dalla variabile LED:

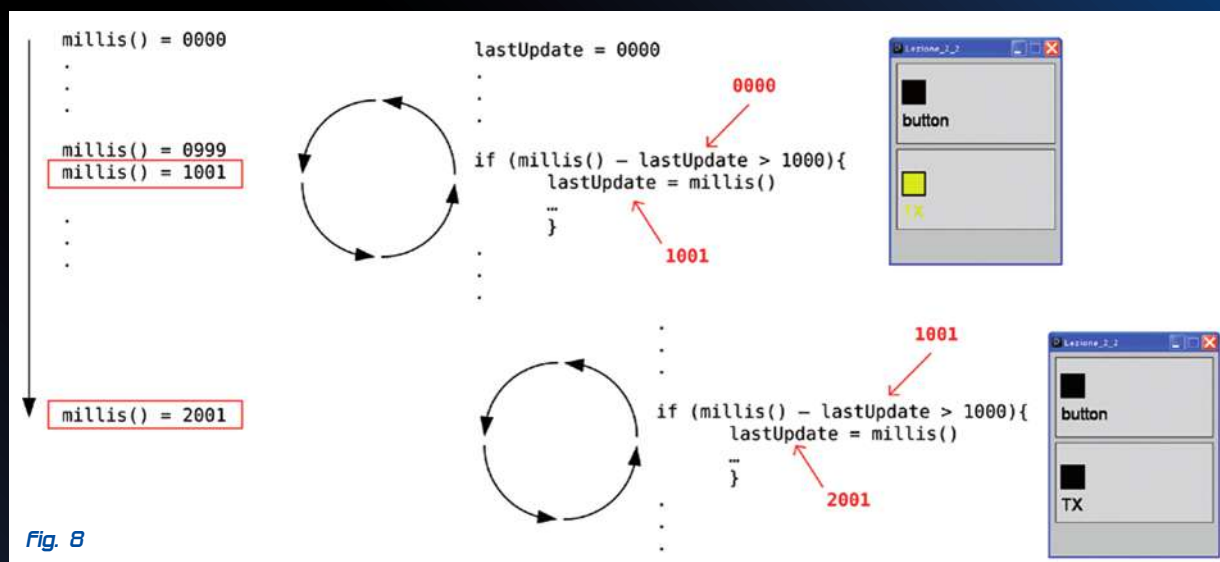


Fig. 8

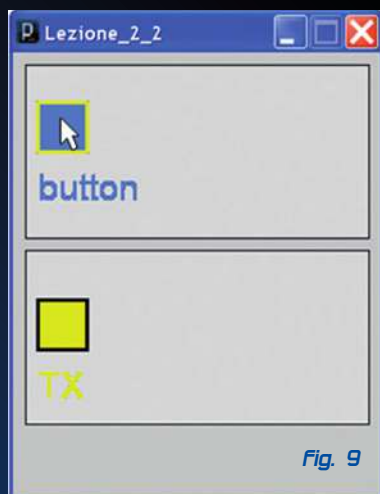


Fig. 9

`fill(241*LED, 250*LED, 18*LED);`
 se `LED = 0`, allora avremo `fill(0,0,0)` cioè riempimento nero, mentre se invece `LED = 1` avremo `fill(241,250,18)` che rappresenta un giallo acceso. Il tutto è mostrato nel codice conte-

nuto nel Listato 3.

Supponiamo ora di voler far lampeggiare il LED in maniera intermittente ogni secondo. La prima cosa che ci viene in mente potrebbe essere inserire un banale ciclo di ritardo all'interno della funzione principale `void draw()`, ma questa tecnica avrebbe un grave effetto collaterale: il pulsante non risponderebbe più durante il ciclo di ritardo, perché il programma sarebbe bloccato dentro questo ciclo e non potrebbe gestire le funzionalità del pulsante.

Il nostro obiettivo è, dunque, far lampeggiare il LED, mantenendo contemporaneamente tutte le funzionalità del pulsante. Una tecnica relativamente semplice per fare ciò consiste nello sfruttare la funzione `millis()` di Processing, soprattutto perché essa funzione restituisce il tempo, espresso in millisecondi, trascorso dall'istante in cui l'applicazione è stata avviata.

Quindi `millis()` ci restituisce un valore che cresce continuamente.

Siccome vogliamo far accadere qualcosa (un evento) ogni secondo (1.000 ms), basterà salvare il valore di `millis()` in una variabile temporanea e confrontare continuamente la differenza tra il valore assoluto di `millis()` ed il valore precedentemente salvato. Se questa differenza è maggiore di 1.000, allora verrà scatenata la relativa azione e sarà nuovamente salvato il valore di `millis()` nella variabile temporanea.

Il ciclo è rappresentato nella Fig. 8. La funzione `millis()` cresce indefinitamente, mentre invece la variabile temporanea `lastUpdate` è inizializzata a 0 ed ogni volta che la differenza tra `millis()` ed essa è maggiore di 1.000, `lastUpdate` viene aggiornata all'attuale valore di `millis()`. Inoltre viene avviata l'azione prevista.

La porzione di codice che svolge quanto descritto è quella contenuta nel Listato 4. La Fig. 9 e la Fig. 7 mostrano come appare il programma definitivo, nel quale il ciclo di lampeggio del LED non interferisce con la funzionalità del pulsante.

LEZIONE 2_3 BUTTON & LED + TRASMISSIONE + ARDUINO

Questo terzo esempio trasforma il programma appena descritto in una vera interfaccia grafica capace di gestire delle funzionalità su Arduino. In pratica vengono aggiunte al

Listato 4

```
void draw(){
  ...

  //ciclo di lampeggio del Led
  //la funzione millis() restituisce il tempo trascorso dall'avvio del programma, in millisecondi
  //se la differenza tra millis() e lastUpdate è > 1000, è trascorso 1 secondo e viene cambiato lo stato di LED

  if ((millis() - lastUpdate) > 1000){
    println("Trasmissione");
    if (LED == 1){
      LED=0;
    }
    else{
      LED=1;
    }
    lastUpdate = millis(); //salvataggio nella variabile lastUpdate del valore di millis()
  }

  ...
}
```

Listato 5

```
import processing.serial.*; //libreria seriale
Serial myPort;             //oggetto myPort costruito dalla classe Serial

int button;                //definizione della variabile button come intero
int LED;                   //definizione della variabile LED come intero
float lastUpdate;

void setup(){
  myPort = new Serial(this, "COM9", 9600); //configurazione della porta seriale, in questo caso è usata
  COM9, deve essere la stessa di Arduino

  ...

}
```

Listato 6

```
//ciclo di lampeggio del Led
if ((millis() - lastUpdate) > 1000){
  println("Trasmissione");
  if (LED == 1){
    LED=0;
  }
  else{
    LED=1;
  }

  if (button == 1){ //se lo stato di button è 1
    myPort.write('H'); //trasmissione sulla seriale del carattere 'H'
  }
  else{
    myPort.write('L'); //se button non è 1, sulla seriale è trasmesso il carattere 'L'
  }

  lastUpdate = millis(); //salvataggio nella variabile lastUpdate del valore di millis()
}
```

programma della lezione 2_2 le primitive di Processing utili per la comunicazione seriale. Allo stesso tempo dovremo anche analizzare il relativo codice di Arduino ed il circuito applicativo.

Per accedere alle funzionalità della comunicazione seriale, dobbiamo far ricorso all'apposita libreria di Processing. Il nostro terzo programma di esempio comincia infatti in questo modo:

```
import processing.serial.*.
```

Ciò significa che inglobiamo nel programma la relativa libreria di comunicazione seriale; dal momento che Processing già dispone di questa libreria non dovremo installare nulla, ma semplicemente richiamarla prima di tutto nel programma. Un'altra operazione fondamentale è nominare la porta seriale:

```
Serial myPort;
```

Serial è una classe e myPort è la definizio-

ne di un oggetto appartenente alla classe Serial, quindi la nostra porta di comunicazione seriale si chiamerà myPort e nel programma si farà riferimento ad essa. Ancora, però, la porta seriale non è funzionante, perché è stata definita unicamente a livello astratto ma non a livello fisico; dobbiamo infatti specificare a quale COM è associata ed a che velocità comunica. Queste definizioni sono fatte dentro void setup(). Il Listato 5 mostra come appaiono nel programma.

La riga di programma seguente:

```
myPort = new Serial(this, "COM9", 9600);
```

significa che myPort (la nostra porta seriale) appartiene alla classe Serial, si chiama 'COM9' e trasmette a 9.600 baud.

Il nome della porta COM che andremo a scrivere ("COM9", "COM6"...) deve essere lo stesso della porta usata da Arduino; ovviamente la velocità di comunicazione deve essere la stessa sia per Arduino che per Processing (nel nostro caso appunto 9600);

Listato 7

```

char val; //dato ricevuto dalla porta seriale
int i;

int LEDPin[6] = {
  5,6,10,11,12,13}; //setup dei pin di uscita

int inPin[6] = {
  2,3,4,7,8,9}; //setup dei pin usati come ingresso

void setup() {
  for (i = 0; i < 6; i++) { // configurazione dei pin di output
    pinMode(LEDPin[i], OUTPUT);
    digitalWrite(LEDPin[i], LOW); // inizializzo l'uscita a 0
  }

  for (i = 0; i < 6; i++) { //configurazione dei pin di input
    pinMode(inPin[i], INPUT);
  }

  Serial.begin(9600); // attivazione della seriale
}

void loop() {

  //stato di lettura e scrittura della seriale

  if (Serial.available()) { // se è disponibile un dato sul buffer seriale
    digitalWrite(LEDPin[0], HIGH); //accendi il LED 0, identificativo dello stato della comunicazione
    val = Serial.read(); //il valore del buffer è letto e salvato su val, una volta letto il buffer è svuotato

    if(digitalRead(inPin[0]) == HIGH){ //lettura dello stato del pin usato come ingresso, se lo stato è alto
      Serial.write(1); //trasmissione sulla seriale del valore 1
    }else{ // altrimenti
      Serial.write(2); //trasmissione del valore 2
    }
  }

  //stato di decodifica del valore seriale ricevuto

  if (val == 'H') { // se il valore di val è il carattere 'H'
    digitalWrite(LEDPin[1], HIGH); // viene acceso il LED 1
  }
  else {
    digitalWrite(LEDPin[1], LOW); // altrimenti LED 1 viene spento
  }

  delay(100); // ciclo di attesa di 100 ms
}

```

se il nome della porta e la velocità non sono rispettati, il sistema non può funzionare. Definiamo ora un semplice protocollo: stabiliamo che se il pulsante è acceso, viene trasmesso verso Arduino il carattere ASCII 'H', altrimenti se il pulsante è spento trasmettiamo il carattere 'L'. Il master della trasmissione è il programma scritto in Processing e la trasmissione del comando deve avvenire ogni secondo. In pratica la struttura è già fatta: infatti sfrutteremo il ciclo di lampeggio del LED descritto nel programma precedente ed aggiungeremo la comunicazione seriale del carattere 'H' oppure 'L'; questa comunica-

zione ovviamente avverrà ogni secondo. Il codice modificato è quello che appare nel Listato 6.

Dentro il ciclo di lampeggio è stata aggiunta una condizione di verifica del valore di button ed a seconda di questo viene scritto in myPort il carattere 'H' oppure 'L', sfruttando la funzione myPort.write().

Ora che il programma master è completo dobbiamo impostare il sistema slave con Arduino. Innanzitutto realizziamo il circuito di Fig. 10 il cui schema elettrico è riportato in Fig. 11; si tratta di collegare due LED con le rispettive resistenze serie ed un pulsante con la sua resistenza di pull-up, ai

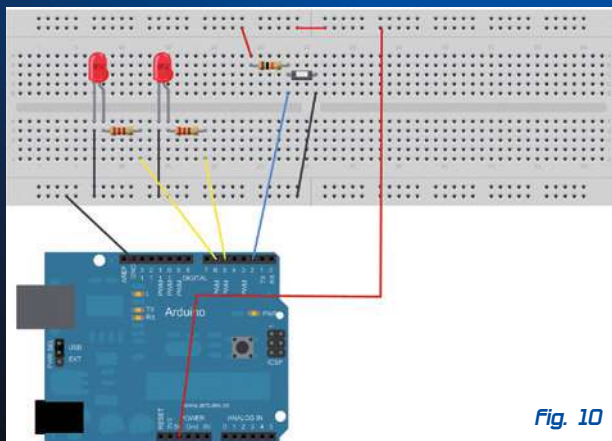


Fig. 10

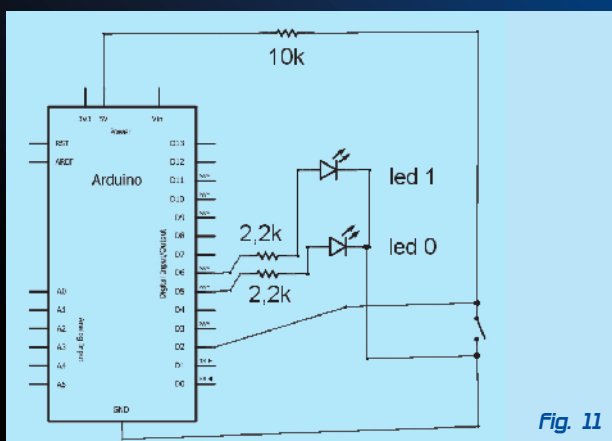


Fig. 11

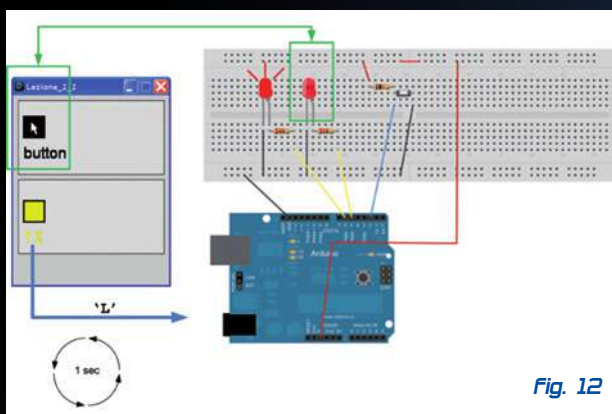


Fig. 12

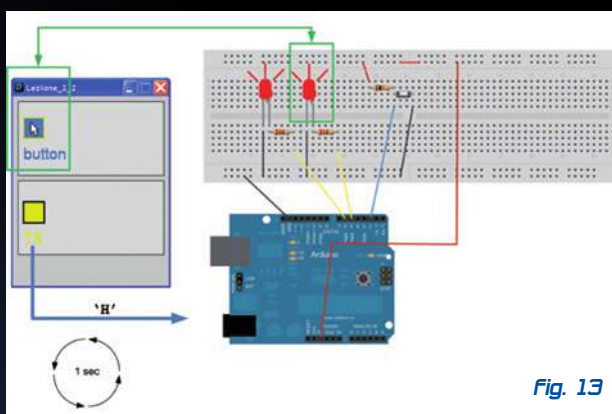


Fig. 13

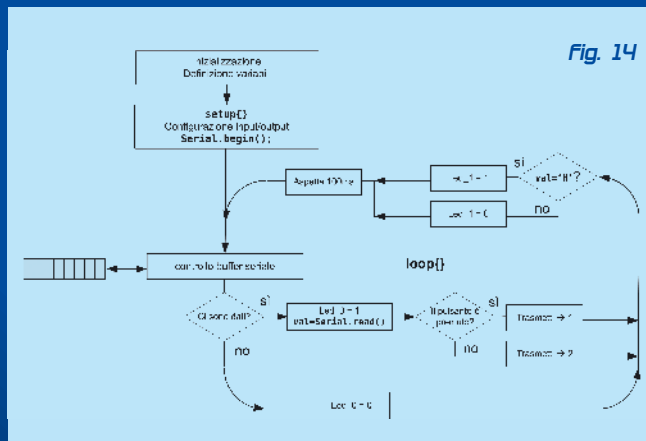


Fig. 14

pin digitali di Arduino che saranno opportunamente configurati come uscite per i LED e ingresso per il pulsante.

I due LED di Arduino saranno chiamati LED_0 e LED_1; il primo servirà per mostrare che la comunicazione seriale è stabilita, quindi sarà acceso se comunque un carattere è ricevuto, mentre LED_1 si accenderà se il carattere ricevuto è 'H' e si spegnerà in corrispondenza del carattere 'L'. Il funzionamento è rappresentato dalle Fig. 12 e Fig. 13. Riportiamo, nel Listato 7, il codice di Arduino.

Per quanto possibile, proviamo a spiegare in questa sede il programma per Arduino: il codice è strutturato in una prima fase in cui sono definiti e configurati i pin di ingresso/uscita e la porta seriale di Arduino, poi il programma entra nel loop principale. Nel loop viene continuamente verificato se ci sono dati nel buffer seriale: in caso affermativo, viene acceso LED_0, quindi letto e salvato il dato del buffer; poi, a seconda dello stato del pulsante, viene trasmesso tramite la porta seriale un valore (1 o 2) verso il PC. Il valore letto dal buffer seriale viene decodificato; se è 'H' viene acceso LED_1, altrimenti lo stesso viene tenuto spento.

Il flow-chart del programma è rappresentato nella Fig. 14. Il programma descritto ha una funzionalità in più rispetto a quella necessaria per il codice di Processing qui descritto, in quanto il programma di Arduino trasmette anche lo stato del pulsante. Questa funzionalità sarà usata dal programma dell'esercizio seguente.

LEZIONE 2_4 BUTTON & LED + TRASMISSIONE/RICEZIONE + ARDUINO
In quest'ultimo programma aggiungiamo

Listato 8

```
//disegno del LedConn
strokeWeight(3);
stroke(0, 0, 0);
fill(247*LEDConn,5*LEDConn,5*LEDConn); //il colore di riempimento del Led dipende dalla variabile LEDConn,
ovvero dal dal suo stato

//se LEDConn=0 ---> fill(0,0,0) quindi colore nero
//se LEDConn=1 ---> fill(247,5, 5) quindi colore rosso

rect(20,350,40,40); //rettangolo che rappresenta il Led
text("RX", 20, 430); //etichetta

if (myPort.available()>0) { // se è presente un dato sul buffer seriale,
  val = myPort.read(); //il dato viene letto e memorizzato su val
  myPort.clear(); //il buffer è ripulito
  if (val == 1){ //se val = 1
    LEDConn = 1; //cambia il valore di LEDConn in 1
  }
  else{ //altrimenti
    LEDConn = 0; //il valore di LEDConn è 0
  }
}
```

sull'interfaccia grafica un ulteriore LED che rappresenta lo stato del pulsante di Arduino. Ora che sappiamo come disegnare un LED ed accenderlo, rimane solo da definire a livello di programma la lettura della porta seriale da parte di Processing. Innanzitutto definiamo un nuovo LED che chiameremo LEDConn; questo, ovviamente, deve essere anche disegnato, perciò procederemo sempre come per il LED precedente: si tratterà di un rettangolo in cui cambieremo il colore di riempimento a seconda del valore di LEDConn. Nel ciclo di trasmissione del master verso Arduino, inseriamo anche la lettura del buffer seriale del PC. Allo scopo, la prima funzione che richiamiamo è `myPort.avaiable()`, la quale restituisce TRUE se c'è qualcosa nel buffer, ovvero FALSE se non c'è nulla. Una volta verificato che il buffer seriale è pieno, questo viene letto ed il valore memorizzato in una variabile temporanea `val`

= `myPort.read()`; dopo aver letto questo valore, il buffer è ripulito attraverso `myPort.clear()`. Verificato che c'è un valore, questo viene letto, poi il buffer è ripulito. Rimane soltanto da decodificare il valore di `val`: se è 1, LEDConn sarà posto ad 1, altrimenti LEDConn sarà posto a 0. La porzione di codice da aggiungere al programma precedente è illustrata nel Listato 8.

Il funzionamento del sistema completo è rappresentato in Fig. 15 e Fig. 16.

NELLE PROSSIME LEZIONI...

Bene, per il momento abbiamo concluso. Nella prossima puntata, seguendo la stessa linea seguita fin qui, descriveremo come avviene la lettura di un valore analogico e spiegheremo il funzionamento della libreria *firmdata*, importante perché ci consentirà di controllare direttamente Arduino da Processing, senza passare per la definizione di alcun protocollo.

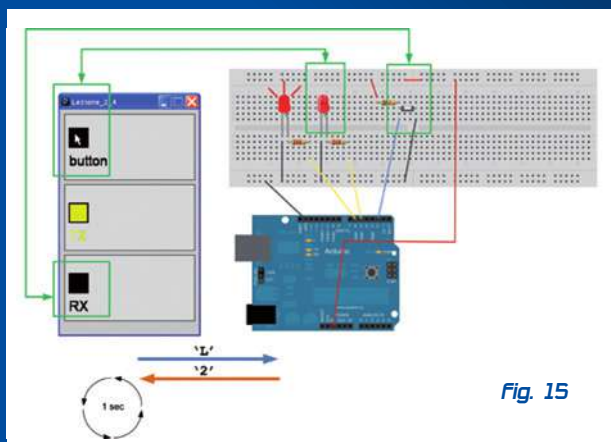


Fig. 15

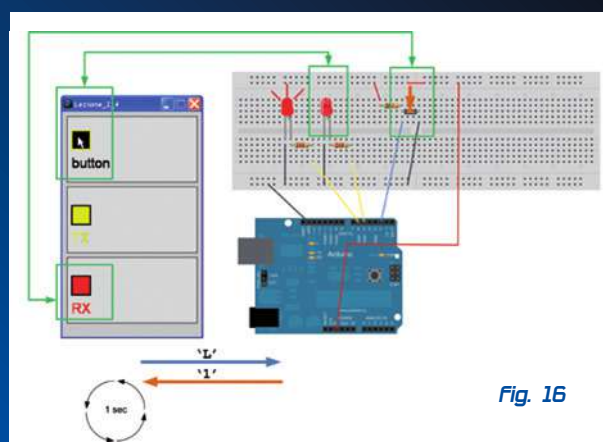


Fig. 16